electronics

MDPI

Article

GvdsSQL: Heterogeneous Database Unified Access Technology for Wide-Area Environments

Jing Shang [1], Limin Xiao [2], Zhihui Wu [1], Jinqian Yang [2], Zhiwen Xiao [1], Jinquan Wang [2,*], Yifei Zhang [1], Xuguang Chen [2], Jibin Wang [1] and Huiyang Li [2]

[1] China Mobile Information Technology Center, Beijing 100840, China; shangjing@chinamobile.com (J.S.); wuzhihui@chinamobile.com (Z.W.); xiaozhiwen@chinamobile.com (Z.X.); zhangyifeiit@chinamobile.com (Y.Z.); wangjibin@chinamobile.com (J.W.)

[2] School of Computer Science and Engineering, Beihang University, Beijing 100191, China; xiaolm@buaa.edu.cn (L.X.); jq.yang@buaa.edu.cn (J.Y.); sy2206251@buaa.edu.cn (X.C.); l11hy@buaa.edu.cn (H.L.)

* Correspondence: derekjqwang@buaa.edu.cn

Abstract: In a wide area environment, leveraging a unified interface for the management of diverse databases is appealing. Nonetheless, variations in access and operation across heterogeneous databases pose challenges in abstracting a unified access model while preserving specific database operations. Simultaneously, intricate deployment and network conditions in wide-area environments create obstacles for forwarding database requests and achieving high-performance access. To address these challenges, this paper introduces a technology for unified access to heterogeneous databases in wide-area environments, termed Global Virtual Data Space SQL (GvdsSQL). Initially, this paper implements a unified data access mechanism for heterogeneous databases through metadata extraction, abstracts the unified access model, and accomplishes identification and forwarding of fundamental database operations. Secondly, the paper introduces a mechanism for expanding database operations through code generation. This mechanism achieves compatibility for special database operations by injecting rules to generate code. Lastly, this paper implements a multilevel caching mechanism for query results in wide-area databases utilizing semantic analysis. Through intelligent analysis of operation statements, it achieves precise management of cache items, enhancing wide-area access performance. The performance is improved by approximately 35% and 240% compared to similar methods.

Keywords: wide-area data management; heterogeneous database; semantic analysis; code generation

Citation: Shang, J.; Xiao, L.; Wu, Z.; Yang, J.; Xiao, Z.; Wang, J.; Zhang, Y.; Chen, X.; Wang, J.; Li, H. GvdsSQL: Heterogeneous Database Unified Access Technology for Wide-Area Environments. *Electronics* **2024**, *13*, 1521. https://doi.org/10.3390/electronics13081521

Academic Editor: Ping-Feng Pai

Received: 7 February 2024
Revised: 31 March 2024
Accepted: 9 April 2024
Published: 17 April 2024

1. Introduction

Technological advancements, including mobile internet, the internet of things, cloud computing, and artificial intelligence, have given rise to novel applications such as smart cities and autonomous driving. These applications generate vast volumes of data, scaling from gigabytes (GB) and terabytes (TB) to the levels of zettabytes (ZB) or even yottabytes (YB) [1]. Data generated by these new applications exhibit two prominent features: wide-area sharing and data heterogeneity. To optimize real-time data processing, service providers frequently establish multiple hierarchical processing clusters. Edge, sub-, or subclusters conduct preprocessing and precomputation, uploading the data to a centralized cluster for further processing. Conversely, the industry has developed databases of multiple types and domains to store data with diverse structures. Examples include relational databases (e.g., MySQL [2], Kingbase [3], and Dameng [4]), document databases (e.g., MongoDB [5]), key-value databases (e.g., Redis [6] and MemCached [7]), and full-text search databases (e.g., ElasticSearch [8]).

Databases frequently utilize distinct data structures and internal implementations for various data types to showcase superior performance in specific domains. Consequently,

each database within a specific domain possesses a unique syntax and access method. These unique access characteristics can pose challenges for higher-tier applications seeking to access the data. High-level applications ideally access databases distributed over a wide area and characterized by heterogeneity through a unified interface. This facilitates wide-area data sharing and unified access to heterogeneous databases, fundamentally reducing the challenges of application development, enhancing resource utilization, and simplifying operational complexity. Presently, numerous scholars have undertaken research on this topic. Song Yao et al. [9–11]. developed a global virtual data space facilitating wide-area file sharing. However, it does not support unified access to heterogeneous databases. Conversely, Zhao Yu et al. [12] devised structured database access technology based on metadata, enabling unified access to various relational databases, yet it does not optimize access to wide-area databases.

Establishing a unified access interface and enhancing the performance of wide-area access are critical for the realization of wide-area heterogeneous database access technology. Significant disparities among modern databases present formidable challenges in implementing wide-area heterogeneous database access techniques. Firstly, heterogeneous databases frequently design distinct data structures and query syntaxes to demonstrate performance advantages in specific domains. This poses challenges in abstracting the access models of each database to achieve unified interface access. Secondly, for operations exhibiting significant differences or special semantics among heterogeneous databases, maintaining their personalized operations based on a unified operation model is of paramount practical significance. Nevertheless, the technology for accessing unified heterogeneous databases in wide-area networks must tackle challenges arising from the complex network environment and deployment scenarios. Firstly, enhancing the access efficiency of wide-area databases is a pressing issue, especially when deploying each database instance in different clusters, attributable to the high latency and unstable characteristics of wide-area networks. Secondly, in wide-area environments, the unified access technology must shield against the complexity of database deployment situations, thereby providing a simple and unified interface for accessing the databases. For achieving automatic localization and forwarding of database access requests, the unified access technology for wide-area heterogeneous databases must implement request resolution and analysis alongside automatic positioning and forwarding of database access requests. This necessitates a request resolution and matching mechanism. In summary, the development of technology for unified access to heterogeneous databases in a wide-area environment entails several challenges:

- There are significant differences in data structures and operational methods between heterogeneous databases, making it difficult to locate and forward database requests to build a unified access layer for heterogeneous databases.
- Heterogeneous databases contain unique management operations, such as node status checks in ElasticSearch, posing challenges in designing an extension mechanism to accommodate unique features of different databases.
- Adding cache in the unified access layer can significantly enhance access performance, yet improving cache effectiveness and reducing the maintenance cost of consistency remains a challenging task.

To address these challenges, this paper proposes a wide-area heterogeneous database access technology, Global Virtual Data Space SQL (GvdsSQL), and implements a prototype system. This technology builds a unified access interface and accelerates cross-domain access through several approaches.

Firstly, this paper proposes a unified data access mechanism for heterogeneous databases based on metadata extraction. It abstracts the database access model to unify the access methods of different databases and achieves precise forwarding and execution of database operations. The main idea of this mechanism is to efficiently match and locate database instances by extracting metadata from databases and operation statements (e.g., table names, field names, and unique identifiers in structured databases). This is achieved by transforming query statements through an abstract database query conversion unit and

then sending the operation statements to the specific database instances, thereby enabling unified access to heterogeneous databases.

Secondly, this paper introduces a code-generation-based database operation expansion mechanism as a supplement to the metadata-extraction-based access mechanism. This mechanism abstracts the system's access process, generating operational code through simple parameter configuration and plugin integration, accommodating custom operations of various databases and enhancing system usability and scalability.

Lastly, this paper implements a semantic-analysis-based multilevel caching mechanism for wide-area database query results to accelerate access and improve system throughput and latency reduction. This mechanism identifies database operations through pattern matching, extracts operation types and scopes, updates query result set caches, and combines access frequency and time-based cache eviction mechanisms with a publish–subscribe model for cache expiration to speed up database access in wide areas.

Extensive experiments were conducted, demonstrating that our method improves performance by approximately 35% and 240% compared to similar methods.

The remaining sections of this paper are organized as follows: Section 2 reviews related work, Section 3 introduces the proposed methods, Section 4 validates the methods through experiments, and Section 5 concludes the paper and looks forward to future research directions.

2. Related Work

2.1. Unified Access Technologies for Heterogeneous Databases

Relational databases, such as MySQL [2], Oracle [13], PgSQL [14], Kingbase [3], and Dameng [4], combine relational data structures, operations, and integrity constraints to offer real-time access, permission protection, and real-time processing. Nonrelational databases are designed for specific scenarios and data types, featuring high-concurrency access, fast retrieval of massive data, and scalability, including key-value, document, and full-text search databases. For instance, Redis [6] (a key-value database) is used for read–write caching, implementing distributed locks, and message queues; MongoDB [5] (a document database) serves as a logging engine, enabling geospatial queries; and ElasticSearch [8] (a full-text search database) is used for document search and data analysis.

Recently, various distributed databases like Spanner [15], TiDB [16], CockroachDB [17], OceanBase [18], and PolarDB-X [19] have emerged, partially supporting wide-area data access. TiDB uses TiKV as its underlying storage engine, aggregating multiple key-value databases and providing SQL interfaces for wide-area data access. CockroachDB, designed for wide-area scenarios, offers robust fault tolerance and high-performance transaction mechanisms, enabling global data sharing and management [17].

In production environments, developers must learn multiple database operations for accessing different types of databases, which leads to issues like data silos, redundancy, and inefficiency in unified access. This hampers the comprehensive utilization of data, reducing application service quality.

To address these challenges, Zhao Yu and others developed a structured database access technology based on metadata, providing unified access to various structured databases and integrating information in distributed environments [12]. Koutroumanis and others developed NoDA, a prototype unified operation layer for nonrelational databases using abstract data models and basic access operators, masking the heterogeneity of nonrelational databases [20]. However, these methods are either conceptual or prototype systems and are not yet practical for use.

The PostgreSQL (PgSQL) team designed Foreign Data Wrappers (FDWs) for unified access to external data, now supporting various relational and nonrelational databases. The Apache Foundation developed Drill [21], providing unified access to heterogeneous data sources like local files, HDFS [22], HBase [23], and MongoDB [5]. Companies like 360 and Tencent developed QuickSQL [24] and SuperSQL [25], respectively, extending Apache Calcite to parse SQL statements and select corresponding data sources and execution

engines, achieving unified access to heterogeneous data sources. However, FDWs are bound to PgSQL and lack flexibility in expansion. Other methods, designed for big data applications, are tied to the Hadoop ecosystem and cannot perform real-time queries. Their focus is on enabling joint queries across heterogeneous databases, with little emphasis on detailed database management. In summary, Table 1 presents a comparison of the advantages and disadvantages of different methods in several dimensions. However, all methods have certain shortcomings when it comes to heterogeneous database unified access technology.

Table 1. Comparisons between existing database access technologies. (The ✓ indicates that the current system has this feature.)

	Zhao Yu et al.'s [12]	**NoDA**	**PgSQL's FDW**	**Apache Drill**	**QuickSQL**	**SuperSQL**
Unified access to structured databases	✓	✓	✓	✓	✓	✓
Unified access to nonrelational databases	✓	✓	Partial support	Partial support	Partial support	Partial support
Provision of fine management features	✓	Partial support	✓	Partial support	Partial support	Partial support
Binding to specific databases or technology	None	None	PgSQL	Hadoop	Hadoop	Hadoop
Ready for deployment	None	Prototype system	✓	✓	✓	✓

2.2. Wide-Area File Caching

In wide-area environments, research on caching is still relatively scarce and is basically file system caching and edge caching. For example, Huo Jiantong and others [26] proposed an edge-caching system in wide area to speed up file access and improve data sharing. Tan and others [27] developed a method in which multiple servers collaborate to provide caching. When clients lack cache elements, requests are redirected to other clients or central servers with the cache, optimizing caching effectiveness and improving access performance. Chen and others [28] introduced a D2D-based caching model, which divides client nodes and servers into clusters and caches replicas in each cluster to improve caching effectiveness. Cloud services such as Amazon [29], Microsoft [30], and Alibaba [31] have developed elastic computing clouds using edge servers to accelerate data access over wide areas.

Relational databases, like MySQL, primarily provide server-side caching from disk to memory. This can only accelerate the query process in the server, while their client-side caching capabilities are limited. As a result, high-cost network access cannot be avoided in wide-area environments, resulting in less effective query caching. Therefore, it cannot be utilized as it should be. Therefore, implementing client-side caching in the unified access layer can effectively enhance wide-area access performance.

3. Design of GvdsSQL

3.1. Overview

As shown in Figure 1, the functions and working methods of each component are as follows:

- Metadata extraction data access mechanism: GvdsSQL unifies the access to all databases as SQL statements. It uses the metadata extraction module to extract the metadata from the SQL statements and from each database (such as the database, data table, fields, etc., in relational databases, and the database, collection, etc., in MongoDB). Subsequently, the metadata matching module performs a matching algorithm to automatically locate the database endpoints that should be accessed. Finally, the data

query execution module converts the SQL statements into corresponding database operations and forwards them to the respective database endpoints.

- Code generation extension mechanism: This mechanism is mainly implemented using the code generation module, which integrates a grammar parser, code snippet library, code generator, and code wrapper. It parses predefined grammars to obtain configuration parameters such as metadata extraction methods and SQL statement conversion methods. It then selects the corresponding snippet from the code snippet library based on the configuration parameters and fills the configuration parameters into the code snippet using the code generator. Finally, it generates the corresponding code package using the code wrapper.
- Semantic analysis multilevel caching mechanism: This mechanism is mainly implemented using the multilevel caching module. The multilevel caching module reuses the capabilities provided by the metadata extraction module and the metadata matching module to identify the types of database operations and the affected cache items and to execute cache invalidation strategies. It comprehensively designs cache eviction strategies based on multilevel cache queues by combining the frequency of data access and the most recent access time. Additionally, this mechanism uses a publish–subscribe pattern instead of the lease mechanism to further reduce the consistency maintenance cost of the cache and enhance performance.

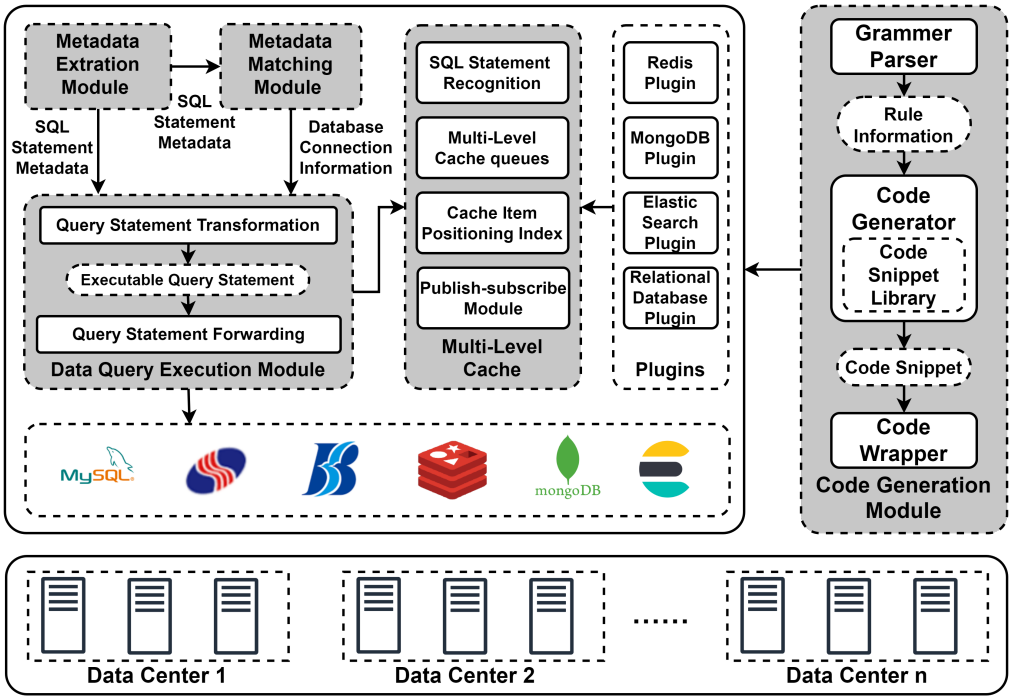

Figure 1. Architecture of GvdsSQL, which is a unified access technology for heterogeneous databases in wide-area environment. GvdsSQL consists of three main components, metadata extraction data access, metadata extraction data access, and semantic analysis multilevel caching.

In summary, the metadata extraction data access mechanism abstracts the access model, achieving unified access to heterogeneous databases. The code generation extension mechanism further extends the capability of unified access to heterogeneous databases, enabling compatibility with specific operations for individual databases. Meanwhile, the semantic analysis multilevel caching mechanism enhances the system's performance in a wide-area environment based on the above two mechanisms.

3.2. The Metadata Extraction Data Access Mechanism

This section provides a detailed introduction to the unified data access mechanism for heterogeneous databases based on metadata extraction. Specifically, it explains the process of extracting metadata from databases and SQL statements, illustrates the working principles of database localization, and discusses the principles behind query statement transformation and execution.

3.2.1. Metadata Extraction

The design of metadata extraction can support data extraction data access mechanisms, code generation extension mechanisms, and semantic analysis multilevel cache mechanisms, providing necessary information for these mechanisms and guiding these modules to make correct decisions. This function consists of database metadata extractors and SQL statement metadata extractors.

The database metadata extractor connects to the corresponding database and exports the metadata of the respective database. For relational databases and document-oriented databases such as MongoDB, which have specific databases or collections describing the database metadata, the database metadata extractor directly exports data from the corresponding tables using the connectors provided by the respective database officials and merges them into the format shown in Table 2. For simple key-value databases (such as Redis), only the database description key-value pair is filled in Table 2, where the key is the database address and the value is empty. For full-text search databases (such as ElasticSearch), access is made to their provided HTTP interface, and multiple interface information is aggregated to merge into the metadata format description, for example, where the index of ElasticSearch corresponds to the entity database description and the type corresponds to the data table description.

Table 2. Description of database metadata format.

Database	Key	Value
Database description (dbs_desc)	Database address (db_address)	Entity database description vector (db_descs)
Entity database description (db_desc)	Table name (table_name)	Table description vector (table_descs)
Table description (table_desc)	Field name (field_name)	Field attribute (field_attr)

By appropriately abstracting and transforming the databases, this method achieves the unified extraction and management of database metadata. However, it is important to note that the structure of the database may change. The method is designed with two mechanisms to ensure the validity of the database metadata. First, it includes a configurable lease time to periodically obtain the metadata description of the database. Second, it incorporates an error feedback mechanism. If the metadata matching locates the corresponding database but the database query execution fails, it will primarily trigger the expiration of the database metadata description and initiate the reacquisition of the database's metadata description.

This method has unified all database accesses into SQL statement accesses. For relational databases, simple adaptation and conversion are sufficient to achieve unified access to the database. However, for nonrelational databases, it is necessary to convert the SQL statements into the specified access format of the corresponding database. Therefore, understanding the information contained in the SQL statements is essential. The SQL statement metadata extractor uses regular expressions to extract the metadata from the SQL statements.

As shown in Table 3, for the query statement in the table, one must obtain field information, table and database information, query condition information, grouping information, paging information, etc. It is worth noting that this method extends the field of table and database information, and users can directly specify the database for executing this query

by specifying the IP address and port. For the update statement, delete statement, and insert statement, the same method is used to extract the metadata contained in the SQL statement. Since these three types of statements contain less information, examples are not provided in this paper.

Table 3. The metadata contained in a select name, count(*) from ' 127.0.0.1:3306'.service.user, where name in (a, b), group by name, and offset 0 size 20.

Type	Value
Field information	name, count(*)
Table & Database information	'127.0.0.1:3306'.service.user
Query condition information	name in (a, b)
Grouping information	group by name
Pagination information	offset 0 size 20

In nonrelational databases, such as MongoDB, the document-oriented overall operation logic is similar to that of relational databases, and the extraction method mentioned above can be directly used. However, for key-value databases and full-text search databases, this method needs to restrict the SQL statements they use. In Tables 4 and 5, the corresponding relationships between operation statements and SQL statements for key-value databases (using Redis as an example) and full-text search databases (using ElasticSearch as an example) are, respectively, displayed. Redis operations are relatively simple and can be followed by the above rules. For ElasticSearch, after parsing the metadata in the SQL statement, parameters can be added to the requested URL for supporting condition queries and grouping operations.

Table 4. Correspondence between key-value database operation statements and SQL operation statements (using Redis as an example).

Operation	Redis Statement	SQL Statement
Query	GET key	select vs. from db0 where k = key
Insert	SET key value	INSERT INTO db0 (k, v) VALUES (key, value);
Modify	SET key value	UPDATE db0 SET vs. = value WHERE k = key;
Delete	DEL key	DELETE FROM db0 WHERE k = key;

Table 5. Correspondence between full-text search database operation statements and SQL operation statements (using ElasticSearch as an example)

Operation	ElasticSearch Statement	SQL Operation Statement
Query Data	`GET /index/type/[id]`	`SELECT type FROM index WHERE id = id;`
Insert Data	`POST /index/type`	`INSERT INTO index (type) VALUES (body);`
Modify Data	`PUT /index/type/[id]`	`UPDATE index SET type = body WHERE id = id;`
Delete Data	`DELETE /index/type/[id]`	`DELETE FROM index WHERE id = id;`

It is worth noting that for performance reasons, this method does not use syntax parsing to obtain the information contained in SQL statements. While this approach supports most database operations, it has the limitation of not being able to support complex SQL parsing. This limitation is addressed using the mechanism described in Section 3.3.

3.2.2. Metadata Matching

As shown in Algorithm 1, we design a metadata matching algorithm to match the database metadata obtained by the metadata extractor with the metadata in the SQL statements in order to accurately locate the database and forward the query request.

Algorithm 1: Metadata Matching Algorithm

```
Input:
  the metadata information of each database E
  the field information of SQL statements Z
  the table and database information of SQL statements T
Output: database connection C
  // If the database address is specified, return the database connection directly.
  if address is not null then
    return connect
  end if
  search = E
  // If it contains entity database information, then filter.
  if meta contains entity info then
    for item in search then do
      // If entity database information cannot be matched, then remove it from the
      search space.
      if item not match meta then
        delete item from search
      end if
    end for
  end if
  // If there is only one match, the connection will be returned.
  if len(search) = 1 then
    return connect
    // If no match is found, an error will be returned.
  else if len(search) = 0 then
    return null
  end if
  // If the table contains information, then filter it.
  if meta contains table info then
    for item in search do
      // If a table does not match, remove it from the search space.
      if item not match meta then
        delete item from search
      end if
    end for
  end if
  // If there is only one match, the connection will be returned.
  if len(search) = 1 then
    return connect
    // If no match is found, an error will be returned.
  else if len(search) = 0 then
    return null
  end if
  // Check if the field information matches.
  for item in search do
    // The data table should contain all the data information.
    for field in Z do
      if item not match field then
        delete item from search
      end if
    end for
  end for
  // If there is only one match, the connection will be returned.
  if len(search) = 1 then
    return connect
  else
    return null
  end if
```

First, as described in Section 3.2.1, this method extends the field of table and database information in the SQL statement. If this field carries specific database connection information, such as the IP address and port information, for example, 127.0.0.1:3306, and the system supports using database aliases to locate the database. It is worth noting that if the system does not include complete database connection information in the metadata, the matching process still needs to be carried out, but it can significantly reduce the search space of the metadata matching algorithm.

Then, according to the standard SQL syntax, the field of table and database information is split, and the elements in this field are analyzed. If it contains physical database description information, all physical database descriptions are aggregated and matched in this search space. If only a unique physical database can be matched, the corresponding database connection is returned from the connection pool. If multiple physical databases are found, then proceed with the matching of table names and field names.

If multiple matching physical databases are found or only table fields are found, search the search space composed of tables. It is worth noting that if multiple matching physical databases are found, then only the search space aggregated by the table information contained in the matched physical databases needs to be searched. If only a unique table can be matched, the corresponding database connection is returned from the connection pool. If multiple tables are found, then continue with the matching of field names.

If multiple matching data tables are found, then search the search space composed of field information. It is worth noting that if multiple matching data tables are found, then only the aggregated search space containing the matched field information of the data tables needs to be searched. If all the field information can be matched in the same data table, the corresponding database connection is returned from the connection pool. If the fields in multiple data tables can match all the field information, the matching fails, and the user is required to specify the database to be connected directly.

The data structure of key-value databases like Redis is relatively simple. Although they include structures such as sets, hashes, and message queues, they can still be abstracted as key-value data types in essence. For this type of data, although it is possible to export all keys as metadata and run a metadata matching algorithm, doing so may incur significant space overhead and consistency maintenance costs. Therefore, when using key-value databases, it is recommended to directly specify the database address to avoid these issues.

It is worth noting that in order to improve the search efficiency of the algorithm, a Trie data structure was used to optimize the search space, reducing the time complexity.

3.2.3. Executing Data Queries

Through the two steps of metadata extraction and metadata matching, this method can accurately locate the database for executing query statements. Next, the SQL statements will be transformed and sent to the specified database for execution, and finally, the returned data will be uniformly processed and wrapped.

In the step of SQL statement transformation, for relational databases, only the different parts of the SQL statement need to be converted. Most operations on data tables do not require special processing, but the management operations of various databases may differ slightly. For example, in the Kingbase database, there is no concept of an entity database. Instead, it uses the concept of "view" to manage a collection of data tables. Therefore, the management of "view" in Kingbase needs to be transformed. As for nonrelational databases, they cannot directly execute SQL statements, so the SQL needs to be converted into specific operation statements.

The method abstracts the query conversion unit, which takes the metadata contained in the SQL statement as input and outputs the corresponding operation method. Specifically, the method abstracts the information contained in the SQL statement, such as field information, table and database information, query condition information, grouping information, pagination information, etc., and combines this information according to the fixed operation methods of various heterogeneous databases to generate complete operational

logic. For example, for relational databases, it converts different keywords and special names. For MongoDB, it converts to the corresponding BSON, and for ElasticSearch, it converts to the corresponding HTTP request by populating URL parameters and request bodies to complete the request.

After completing the transformation from SQL statements to operation statements, the method will execute the query, obtain the corresponding database connection, and forward the data to the corresponding database. After the database returns the execution result, the method will uniformly wrap the data returned by various databases for consistent upper-layer operations. First, it preserves the original return information. Second, for query requests, it extracts the corresponding field information and field descriptions, serializes the data into object arrays corresponding to the fields, and caches the data, as detailed in Section 3.4. Third, for other requests causing data changes, it returns the type of operation and the scope of the operation's impact, invalidates the cache, and notifies other nodes, as detailed in Section 3.4.

At this point, the method can basically complete the unified access to heterogeneous databases. However, each database has specific operations that cannot be completely abstracted, so the solution described in Section 3.3 is designed.

3.3. Code Generation Extension Mechanism

This section provides a detailed introduction to the database operation extension mechanism based on code generation. Firstly, it introduces the interface definition for code generation extension, and then it explains the overall process of code generation.

3.3.1. Interface Definition

To accommodate the special operations of various databases, this method uses an extension mechanism we designed based on code generation. In order to achieve unified access extension for a heterogeneous database, it is necessary to implement SQL statement recognition and conversion rules. Specifically, the SQL statement recognition rule can be a string or a regular expression. If the SQL statement sent by the user successfully matches the SQL statement recognition rule, this method will execute the rules designed in the code generation extension. The conversion rule is a function definition that takes the extracted metadata information as a parameter and converts these metadata into the query statement of the specified database. Optional SQL metadata extraction rules can also be defined. If this rule is not set, the metadata extraction rule described in Section 3.2.1 will be used.

For example, Table 6 demonstrates how to implement the node status viewing function in ElasticSearch through the code generation extension mechanism. In this example, the regular expression "select * from (.)n̈ode_status where local='(.)';" is used to match the SQL statement submitted by the user. If a successful match is made, the user's custom metadata extraction and conversion rules will be executed. The metadata extraction rule only focuses on field information, table and database information, and query condition information. The SQL statement conversion rule then converts the extracted metadata information into the specified HTTP request, i.e., sending a request to the "/_nodes/*" URL. It is worth noting that when the code generation extension is integrated into the system, the SQL statement matching rule will be executed with priority to ensure the correctness of the conversion and recognition. Additionally, in order to improve the performance of this matching process, this method utilizes a multithreading mechanism for optimization.

3.3.2. Code Generation Process

In Figure 2, the main process of code generation is demonstrated. The code generation primarily consists of a grammar parser, code snippet library, code generator, and code wrapper. Initially, the system calls the grammar parser to format the user input parameters, extracting the SQL matching rules, conversion rules, and metadata extraction rules. Subsequently, the parameters are passed to the code generator unit, which selects the corresponding code snippets from the code snippet library based on the number and content of

the parameters, and then fills the parameters into the corresponding code snippets. Finally, the code wrapper is used to combine the separated code snippets into a complete integrable code generation extension.

Table 6. Example of ElasticSearch query node status information code generation extension.

Name	Rule
SQL statement recognition rule	`select * from '127.0.0.1:9200'.node_status where local='_local';`
SQL statement transformation rule	`GET http://127.0.0.1:9200/_nodes/_local`
Metadata extraction rule	`<Field Information, *>,<Table and Database Information, node_status>,<Query Condition Information, local='_local'>`

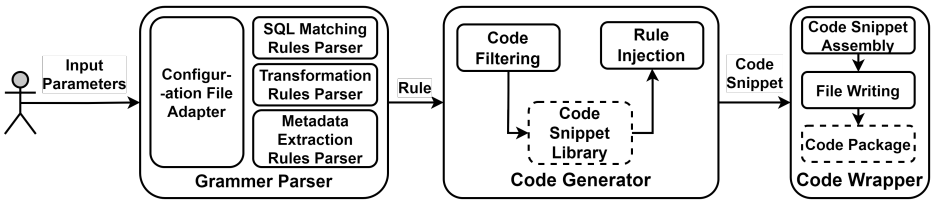

Figure 2. Overview of the code generation process. First, user input is formatted, and appropriate code snippets are filtered from the code database. Then, a code wrapper is used to complete the code packaging.

Specifically, the grammar parser formats the user input parameters according to the interface definition in Section 3.3.1. Through reasonable abstraction, the grammar parser of this method can support various file formats such as JSON, YAML, JAVA, etc., and extract the required rule information from these file formats. After extracting the rule information, the grammar parser will validate the legality of the parameters, such as excluding special characters, validating parameter formats, and validating the legality of conversion rules. The code snippet library integrates all templated code and splits it based on function granularity to allow flexible combinations, ensuring the readability and conciseness of the code. The code generator receives the parsing results from the grammar parser and filters code snippets based on the parsing results, such as selecting different code snippets for code conversion rules for Redis and MongoDB and for whether the metadata extraction rule is passed. Then, the code generation unit fills the parameters from the grammar parser into the specified code snippets. After the code generation unit fills the data, the generated code is still not executable. Finally, these codes are passed to the code wrapper, which assembles the code snippets into a complete runnable project, primarily injecting necessary build tools, creating a complete file structure, and generating corresponding documents.

The generated extension is packaged into a complete package and is dynamically loaded and run in real time during the program's execution, enhancing the overall availability and scalability of the system. As the metadata extraction data access mechanism currently only covers a portion of database operations and cannot provide complete database operation capabilities, the system needs to provide convenient and simple extension capabilities. This method uses a code generation approach to avoid a large amount of repetitive work, ensures code quality, and allows users to focus on specific implementation codes without needing to learn complex plugin-writing methods. However, it is important to note that the code generation extension is just one extension solution, as most operations can be completed through the metadata extraction data access mechanism.

On the other hand, this method recognizes the enormous potential of the code generation approach. It significantly reduces the user's access difficulty by generating a complete project from specified SQL statements, specifically by replacing the three types of rules in the interface definition with executable statements for a specified database while also providing field-level access control capabilities for relational databases by extracting metadata from SQL statements.

3.4. Semantic Analysis Multilevel Cache Mechanism

This section provides a detailed introduction to the multilevel cache mechanism for wide-area database query results based on semantic analysis. The section first introduces the overall structure of the semantic analysis multilevel cache mechanism, then discusses the cache management mechanism, and finally addresses the cache consistency issues in a wide-area environment.

3.4.1. Infrastructure

In Figure 3, the overall structure of the semantic analysis multilevel cache system is displayed. The system includes an SQL statement recognition module, multilevel cache queues, cache item positioning index, and publish–subscribe module.

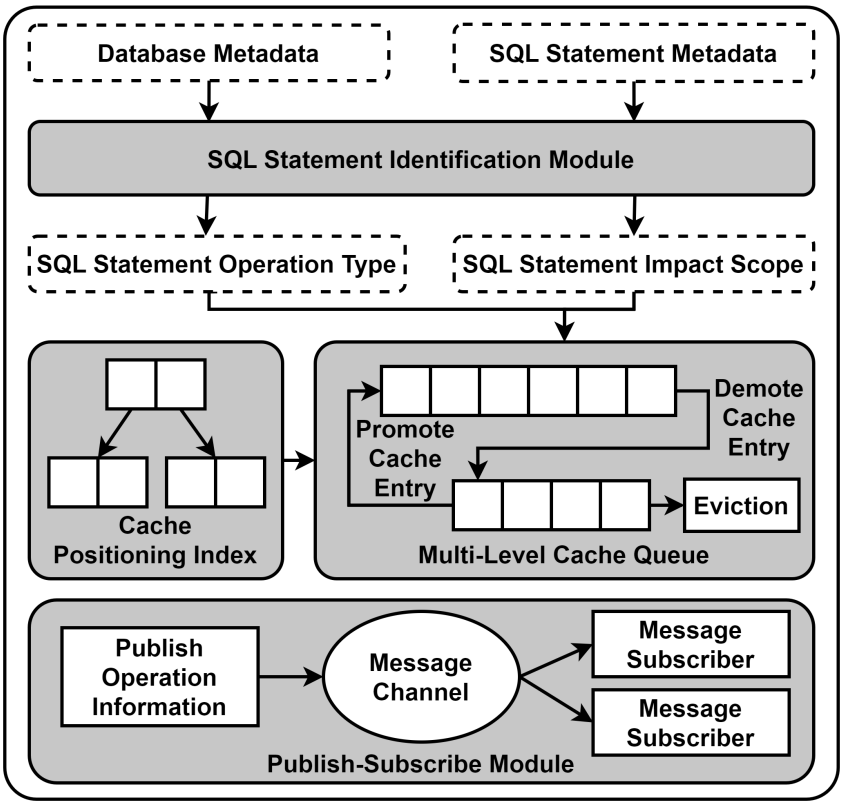

Figure 3. Overview of semantic analysis multilevel cache system. It includes an SQL statement recognition module, multilevel cache queues, cache item positioning index, and publish–subscribe module.

The SQL statement recognition module receives the metadata of the database and SQL statements as parameters, identifying the type and scope of the query statements to be executed. Specifically, the SQL language recognition module can obtain the type of request, field information, table and database information, conditional query information, grouping information, pagination information, etc., through metadata extraction, and it locates the database connection required for data access through metadata matching. The SQL language recognition module analyzes the statement type, executes the logic for setting the cache if it is a query statement, and executes the logic for deleting the cache otherwise. When deleting the cache, the SQL language recognition module will calculate the scope affected by the corresponding statement and delete the cache within the smallest scope. For example, if the execution results of "select name from user" and "select info from user" are already cached in the cache, when executing the statement "update user set name = foo", only the cache result of the first SQL statement needs to be deleted. For a more detailed discussion, refer to Section 3.4.2.

The multilevel cache queue module stores specific cache items, manages the cache based on the recognition results of the SQL language recognition module, and runs background tasks to periodically recycle cache items. Specifically, the multilevel cache queue module contains at least two cache queues, numbered from 1 to n in descending order of priority. This method places the cache item in queue 1. Subsequently, the multilevel cache queue takes over the management of cache items. Cache management is divided into two parts: First, data modification leads to cache expiration, which can be further classified into the SQL language recognition module, pushing invalidation information, and remote services, pushing invalidation information through the publish–subscribe module. Second, it manages cache items based on data access frequency and access time, degrading cache items between queues until elimination or promoting cache items between queues until queue 1 is reached; for a more detailed discussion, refer to Section 3.4.2.

Traversing a multilevel queue access cache is extremely time-consuming, so this method designs a cache item positioning index to accelerate cache access. The multilevel queue contains a data structure composed of a hash table and a prefix tree, where the hash table is divided into three layers. As shown in Figure 4, the keys of each layer of the hash table are sequentially the database name, entity database name, and table name. The value of each layer of the hash table except the last layer is the next layer of the hash table. The value of the last layer is a prefix tree composed of data table field names. When caching data, the cache item positioning index will sort the fields and build a prefix tree according to this order. For example, when accessing the cached result of "select name, info from 127.0.0.1:3306.db.user", it will first query all entity databases in 127.0.0.1:3306, then query all tables in the database, and then query the corresponding prefix tree according to user. Finally, it will sort the fields, obtain the sequence of "info" and "name", and access the prefix tree to query cache items in this order. It is worth noting that when using a key-value database like Redis, this method will inject fake database names and entity database names to ensure the uniformity of the data structure.

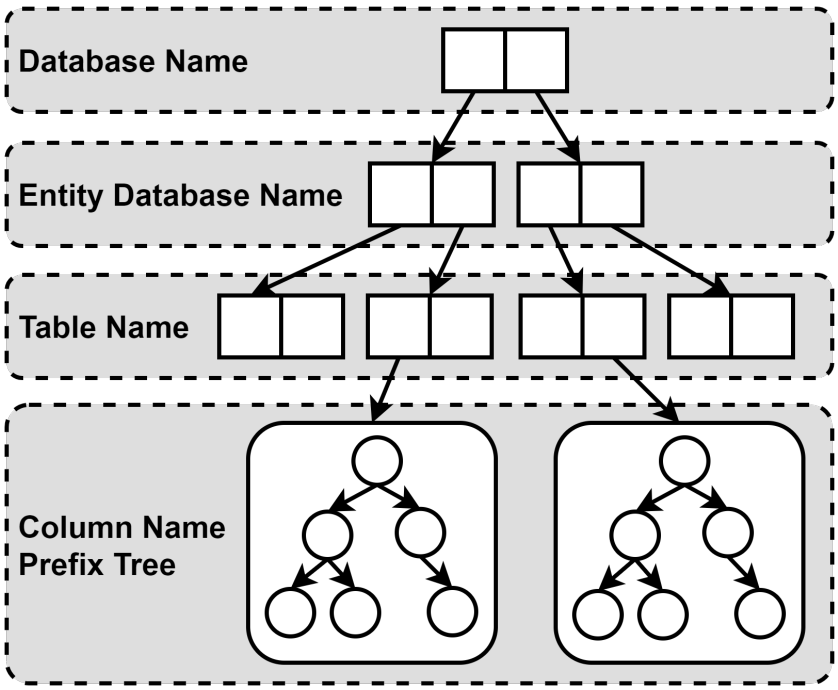

Figure 4. Overview of multilevel cache which includes four levels: database name, entity database name, table name, and column name prefix tree.

The publish–subscribe module is responsible for communicating with systems deployed in other data centers and maintaining cache consistency. The publish–subscribe

module receives messages at the granularity of a table, and when the SQL statement recognition module identifies a data query, it will pass the table information to the publish–subscribe module, which will then subscribe to modifications from other data centers based on the table information. When the SQL statement recognition module identifies a data modification, it will pass the table information and SQL statement to the publish–subscribe module, which will then push the SQL statement to the subscribers based on the table information. For example, if the current program is deployed in two data centers, when data center A executes "select name from user", it will cache the statement execution result and subscribe to modifications about the "user" table from data center B. When data center B executes the statement "update user set name = foo", it will push this SQL statement to data center A, and the cached result in data center A will expire.

3.4.2. Cache Strategy

When managing cache items, this method is divided into two parts: If the data involved in the cache are modified and the data in the cache are no longer the latest, they need to be actively deleted. The capacity of the cache is limited, and it needs to periodically clean up data that are not frequently accessed in the cache. When a request is sent, the SQL statement undergoes two processes: metadata extraction and metadata matching (as described in Sections 3.2.1 and 3.2.2). Metadata extraction can obtain the type of request, field information, table and database information, conditional query information, grouping information, pagination information, etc. Metadata matching can locate the database connection that needs to execute the data access. Then, the type of SQL statement is identified. If it is a query statement, no cache expiration processing is performed. If it is a delete, update, or insert statement, the impact range of the SQL statement is further judged. When the SQL statement type is delete or insert, this method uses the cache item positioning index to find the possibly affected cache items at the granularity of the table and clears all cache items. For example, if the cache contains the execution results of "select name from user" and "select info from user", when the statement "insert user value(foo, bar)" is executed, all cached results of the "user" table are deleted. When the SQL statement type is updated, this method uses the cache item positioning index to find the possibly affected cache items at the granularity of the field and further compares the query conditions to minimize the number of deleted cache items. For example, if the cache contains the execution results of "select info from user", "select name from user where id = 1", and "select name from user where id = 2", when the statement "update user set name = foo where id = 2" is executed, only the execution result of "select name from user where id = 2" is deleted.

On the other hand, this method ensures the validity of the cache by evaluating the access frequency and access time of cache items among the various queues. Firstly, this method defines an access bit and an access buffer in each cache item. If the access bit of a cache item is accessed within the sampling time, the access bit will be set to 1, otherwise it will be set to 0. The size of the access buffer is set to 16, used to store the access bit in the past 16 sampling periods. Secondly, this method defines a periodic task to periodically check the access bit of cache items. The task will move the value of the access bit to the end of the access buffer and delete the oldest access record.

By recording the values in the access buffer, this method uses the following formula to evaluate the access situation of cache items:

$$s = \sum_{i=0}^{15} k_i \cdot 2^{\lfloor i/4 \rfloor}$$

where s represents the evaluation result of the cache item, i represents the index of accessing data in the cache, and ki represents the value accessed in the cache, which can only be 0 or 1. The above formula explains that this method groups the data in the cache into groups of four, and the data in the same group have the same weight value, and the latest access record has the highest weight. Through this formula, this method unifies the evaluation of

access frequency and access time, comprehensively considering the likelihood of future access to cache items. When the evaluation result s is the same, this method considers that the cache item with access time closer to the present is more important. For example, if the access sequences of two cache items are "0000000000000010" and "0000000000000001", their evaluation results are the same, but the second cache item is more important.

In this method, each level of the queue is set to be 1.5 times the size of the next set queue, and the total number of cache items in all queues is specified, so the cache capacity is limited. This method sets a water level for the cache capacity. When the data of cache items exceed the water level, this method will recycle the cache space. Specifically, the system will first calculate the number of cache items that need to be removed from the cache and then remove the corresponding number of cache items in the last-level cache. Then, the cache is gradually reduced to ensure that the number of cache items in each level of the queue is 1.5 times the size of the next level.

When the evaluation result of a cache item is recalculated and is greater than the original value, this method compares the relationship between its evaluation value and the minimum evaluation value in the previous level queue. If the current evaluation value of the cache item is greater, then the cache item is promoted to the previous level queue.

4. Experiment and Verification

4.1. Experiment Environment

In this paper, an experiment environment was set up using three servers in two different locations. The server configurations are shown in Table 7. All servers were equipped with Intel Xeon 4114 processors, 64 GB memory, 512 GB SSD, and 4TB HDD, running on CentOS 7.8 operating system . The connection bandwidth between the two locations was measured using speed-testing software, resulting in 13.88 MB/s.

Table 7. Server configuration.

Configuration Item	Configuration Information
CPU	Intel(R) Xeon(R) Silver 4114 CPU @ 2.20 GHz
Memory	64 GB
Disk	512 GB SSD, 4T HDD
Operating system	CentOS 7.8

Table 8 displays the heterogeneous databases and their versions used for testing in this paper. Four common database types were selected based on the DB-engine website [32]: relational databases, document databases, KV databases, and full-text search databases. The highest ranked databases for each type were then selected based on the website rankings. These databases included MySQL 5.7.37, MongoDB 6.0.4, Redis 7.0.8, and ES 7.17.5. This decision was made due to licensing issues with the most popular relational database, Oracle. For compatibility and testing purposes, we chose Dameng 8.1.1.126 and Kingbase V8 R6, two Oracle-compatible databases.

Table 8. Heterogeneous database versions used in testing.

Software	Version
MySQL	MySQL 5.7.37
Kingbase	KingbaseES V8 R6
Dameng	DM 8.1.1.126
Redis	Redis 7.0.8
MongoDB	MongoDB 6.0.4
ElasticSearch	ElasticSearch 7.17.5

This paper selected the FDW method based on PgSQL and QuickSQL, an open-source tool from 360 companies, as a comparative evaluation of the performance of this method. FDW, based on PgSQL, implements the capability to uniformly access external data and

currently supports various relational and nonrelational databases. QuickSQL extends Apache Calcite, selecting the corresponding data source and execution engine by parsing SQL statements, achieving unified access to heterogeneous data sources.

4.2. Functional Testing

GvdsSQL was implemented by using 14,896 lines of Java code, which include a metadata extraction data access mechanism, code generation extension mechanism, and semantic analysis multilevel caching mechanism. Additionally, it includes some management APIs for use on the administration page. Currently, GvdsSQL supports wide-ranging unified access to three relational databases: MySQL, Kingbase, and Dameng (with Kingbase and Dameng being domestically developed databases), as well as three nonrelational databases: Redis, MongoDB, and ElasticSearch. JDBC is used for accessing relational databases, while Jedis, the MongoDB official connector, and Okhttp are employed for accessing ElasticSearch. Additionally, to reduce the maintenance cost of database connections, this paper introduced a database connection pool that manages database connections and provides connection reuse functionality. In GvdsSQL, all parts except for the semantic analysis multilevel cache mechanism are stateless applications that can be deployed and expanded at any location, providing better scalability. However, the cache entries in the semantic analysis multilevel cache mechanism only adopt the semantic analysis expiration policy and fixed time expiration policy. This may result in limitations, as the cache items deployed in different locations may be inconsistent. We aim to improve this in the future.

On the other hand, as this method requires extracting metadata from databases, prior knowledge of the database connection information is necessary. Therefore, a frontend interface for managing database connection information was designed, as shown in Figure 5.

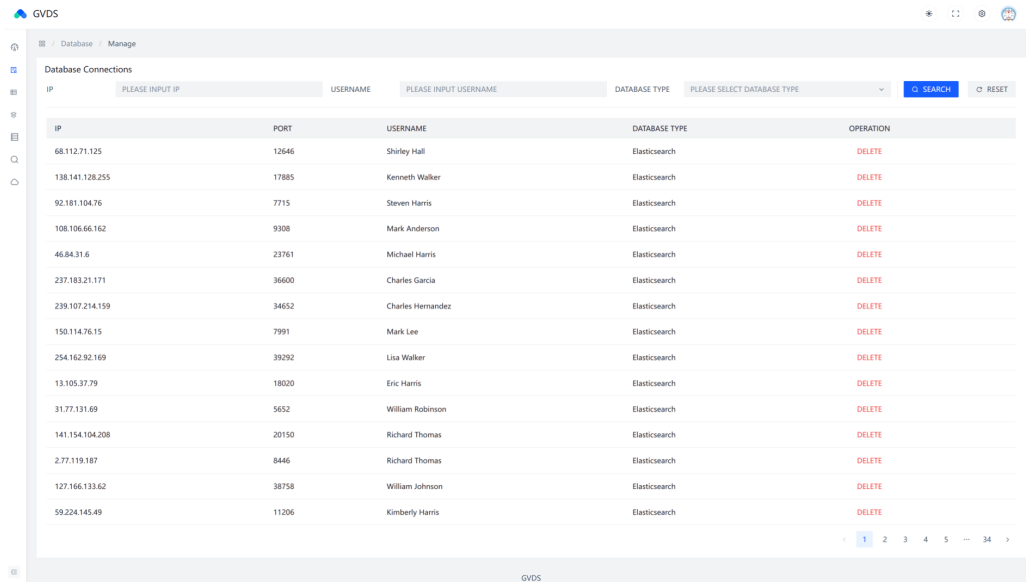

Figure 5. Database connection information management.

Moreover, the frontend page integrates features related to database visualization management and code generation, as depicted in Figure 6. Users can directly manage and visualize data in various databases through the interface. Additionally, as shown in Figure 7, users can invoke the code generation module to generate system extension plugins.

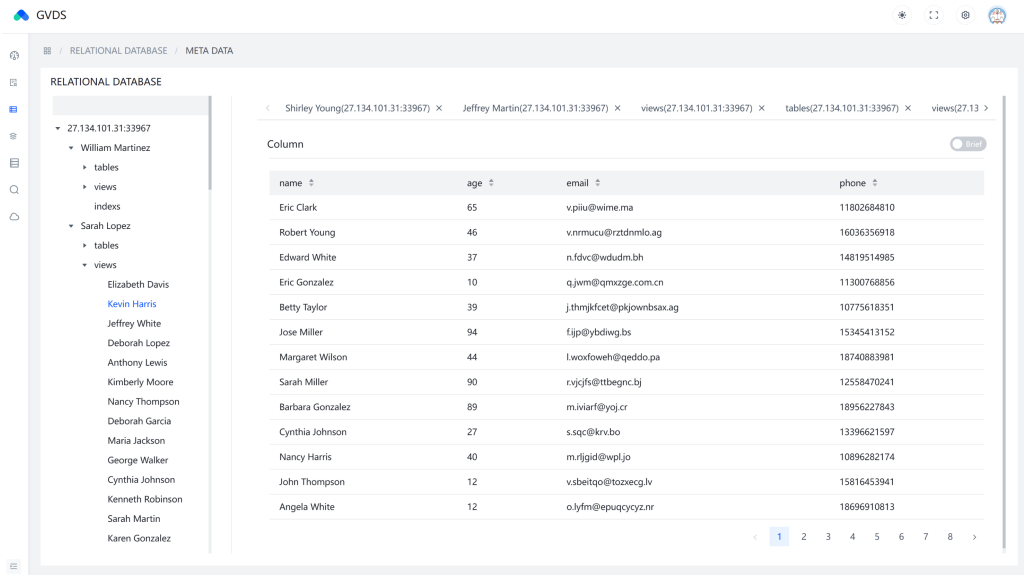

Figure 6. Database information visualization.

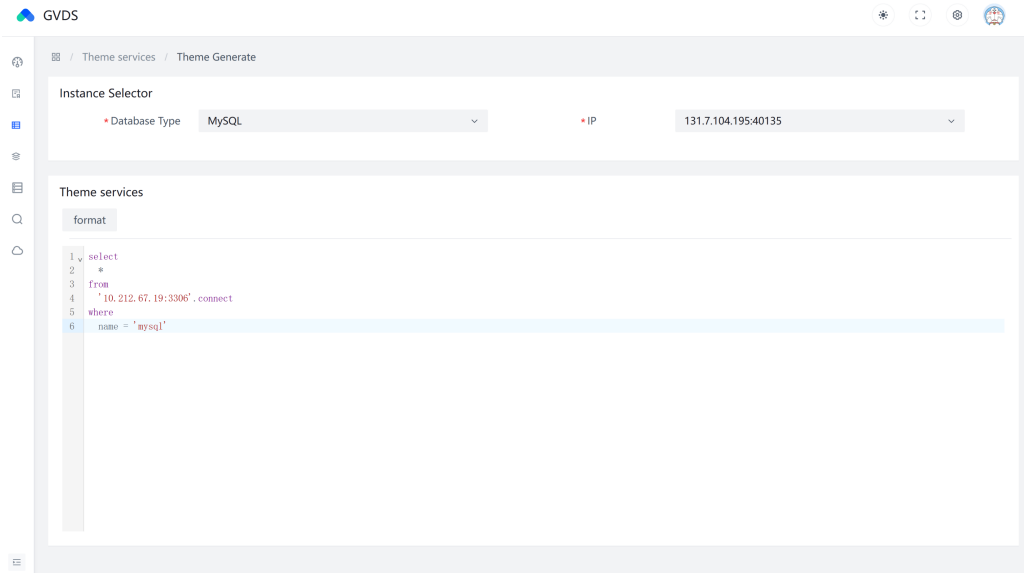

Figure 7. System extension plugin generation.

4.3. Performance Testing

In this section, the paper compares the performance differences between GvdsSQL, FDW, and QuickSQL in terms of throughput and operation latency. It is noteworthy that GvdsSQL achieves unified access to MySQL, Kingbase, Dameng, Redis, MongoDB, and ElasticSearch. FDW provides unified access to MySQL, Redis, and MongoDB, while QuickSQL achieves unified access to MySQL, Hive, Hbase, and ElasticSearch. Therefore, this section only compares the performance of the three methods in operating MySQL databases. The GvdsSQL prototype system was deployed on a server with the CentOS 7.6 operating system. Then we created a table with one million rows in a MySQL database. Test scripts were written using JMeter [33], and 32 worker threads were opened for performance testing.

Figure 8 depicts the throughput comparison of GvdsSQL, FDW, and QuickSQL. The throughput of GvdsSQL is 310, 279, 283, and 270 for query, modify, insert, and delete, respectively. When querying data, GvdsSQL's throughput is 32.47% higher than FDW and 520% higher than QuickSQL. When modifying data, GvdsSQL's throughput is 43.81%

higher than FDW. When inserting data, GvdsSQL's throughput is 48.94% higher than FDW. When deleting data, GvdsSQL's throughput is 37.05% higher than FDW.

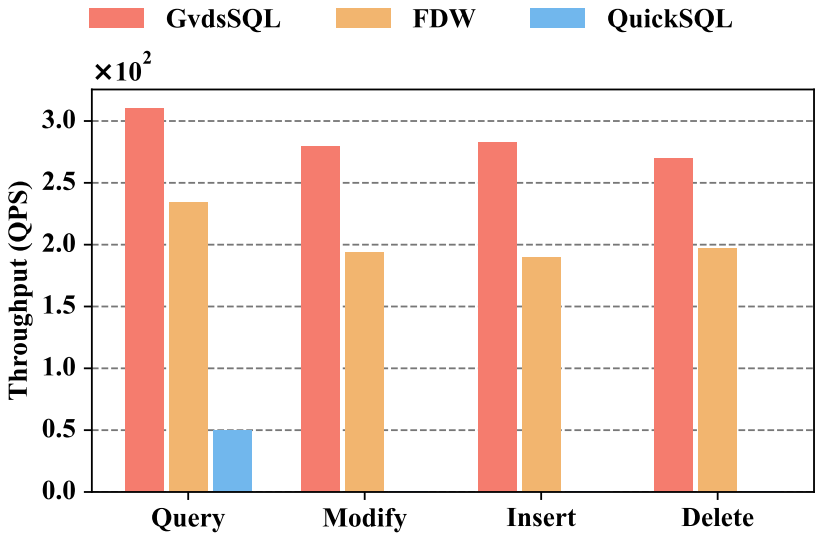

Figure 8. Throughput comparison of GvdsSQL, FDW, and QuickSQL.

Figure 9 illustrates the operation latency comparison of GvdsSQL, FDW, and QuickSQL. The operation latency of GvdsSQL is 3.23 ms, 3.58 ms, 3.53 ms, and 3.70 ms for query, modify, insert, and delete, respectively. When querying data, GvdsSQL's latency is 24.51% lower than FDW and 83% lower than QuickSQL. When modifying data, GvdsSQL's latency is 30.46% lower than FDW. When inserting data, GvdsSQL's latency is 32.86% lower than FDW. When deleting data, GvdsSQL's latency is 27.03% lower than FDW.

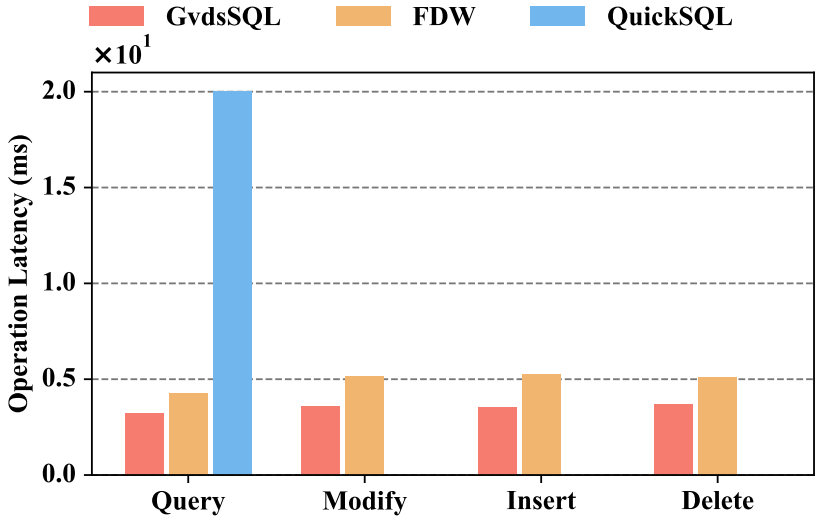

Figure 9. Operation latency comparison of GvdsSQL, FDW, and QuickSQL.

It can be observed that GvdsSQL has higher throughput and lower latency than the other two solutions. This is mainly because GvdsSQL uses simple regular expressions to extract metadata from SQL statements and designs a fast database matching mechanism to achieve the localization and forwarding of database requests. FDW is coupled with PgSQL, requiring requests to be forwarded to the PgSQL database first for syntax parsing and then forwarding the corresponding data to the corresponding operation node. For QuickSQL, although database operation requests do not need to go through two layers of forwarding, there is a performance gap in SQL statement parsing speed and direct

information extraction using regular expressions. Additionally, QuickSQL's more complex composition and implementation of additional functionalities contribute to a certain performance decline. Furthermore, in GvdsSQL and FDW, the throughput and latency differences between querying and modifying, inserting, and deleting operations are not significant. This is because the main factor limiting performance in a wide-area environment is the unstable network conditions, and the performance difference in database operations itself is not obvious.

Figure 10 presents the comparison between GvdsSQL and FDW under different write operation ratios. When including 25% write operations, GvdsSQL's operation latency is 5.56 ms, 13.93% lower than FDW. When including 50% write operations, GvdsSQL's operation latency is 7.01 ms, 22.40% lower than FDW. When including 75% write operations, GvdsSQL's operation latency is 10.02 ms, 32.25% lower than FDW.

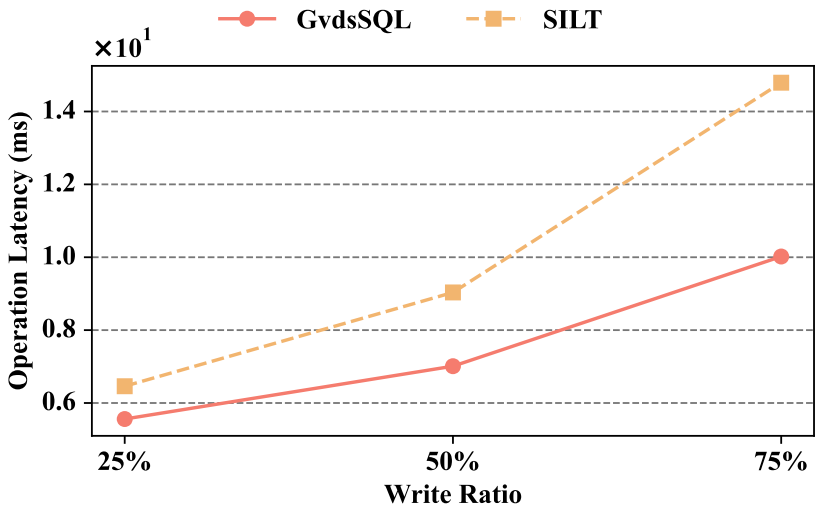

Figure 10. Operation latency comparison of GvdsSQL and FDW under different write operation ratios.

It can be observed that the latency of all methods increases with the rise in write operation ratio, and GvdsSQL outperforms FDW under different write operation ratios. This is mainly because GvdsSQL provides an effective caching mechanism through a semantic-analysis-based multilevel caching mechanism, accurately locating the range of cache items affected by database operation statements and controlling the number of invalidated caches within a certain range, significantly reducing the number of database queries. For FDW, as it abstracts heterogeneous data sources as data and data tables, its caching strategy aligns with the database-query-caching strategy. When a data table is modified, all related cache items will expire, and the invalidated cache cannot be restricted within a certain range.

5. Conclusions

This paper discusses a solution proposed for unified access to heterogeneous databases in a wide-area environment. The proposed solution includes three mechanisms: a unified data access mechanism based on metadata extraction, an extension mechanism based on code generation, and a multilevel caching mechanism based on semantic analysis. A complete prototype system was implemented. The method addresses issues related to data sharing and management in wide-area environments, aggregating heterogeneous databases and mitigating problems such as database isolation, data redundancy, and complex maintenance. Experimental validation confirmed the correctness and superior performance of the proposed method.

In the future, the research will focus on areas such as distributed transactions among heterogeneous databases, further improving system availability, and reducing the complex-

ity of system integration. The goal is to develop a comprehensive and production-ready framework and standards for unified access to wide-area heterogeneous databases.

Author Contributions: Methodology, J.S., L.X., Z.X. and Y.Z.; Software, J.Y., J.W. (Jinquan Wang), Y.Z., J.W. (Jibin Wang) and H.L.; Validation, Z.W.; Formal analysis, J.S. and Z.W.; Investigation, Z.X.; Resources, J.S. and Z.X.; Data curation, Z.W.; Writing—original draft, J.Y. and J.W. (Jinquan Wang); Writing—review & editing, X.C. and H.L.; Visualization, X.C.; Project administration, L.X. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National Natural Science Foundation of China under Grant U23B2027, China Mobile Joint R&D Project under Grant R2310DDY.

Data Availability Statement: The data presented in this study are available on request from the corresponding author due to that GvdsSQL is a commercial project.

Conflicts of Interest: Authors Jing Shang, Zhihui Wu, Zhiwen Xiao, Yifei Zhang and Jibin Wang were employed by the company China Mobile Information Technology Center. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

References

1. Qian, D.; Wang, R. Key issues in exascale computing. *Sci. China Inf. Sci.* **2020**, *50*, 1303–1326.
2. MySQL rev.5.7.37. 2022. Available online: https://www.mysql.com/ (accessed on 2 March 2023).
3. KingbaseES rev.8.6.7. 2022. Available online: https://www.kingbase.com.cn/ (accessed on 2 March 2023).
4. Dameng rev.8.1.1.126. 2023. Available online: https://www.dameng.com/ (accessed on 2 March 2023).
5. MongoDB rev.6.0.4. 2023. Available online: https://www.mongodb.com/ (accessed on 2 March 2023).
6. Redis rev.7.0.8. 2023. Available online: https://redis.io/ (accessed on 2 March 2023).
7. Nishtala, R.; Fugal, H.; Grimm, S.; Kwiatkowski, M.; Lee, H.; Li, H.C.; McElroy, R.; Paleczny, M.; Peek, D.; Saab, P.; et al. Scaling memcache at facebook. In Proceedings of the Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), Lombard, IL, USA, 2–5 April 2013; pp. 385–398.
8. Elasticsearch rev.7.17.5. 2023. Available online: https://www.elastic.co/ (accessed on 2 March 2023).
9. Xiao, L.M.; Song, Y.; Qin, G.J.; Zhou, H.J.; Wang, C.B.; Wei, B.; Wei, W.; Huo, Z.S. GVDS: A Global Virtual Data Space for Wide-area High-performance Computing Environments. *Big Data Res.* **2021**, *7*, 123–146.
10. Qin, G.; Xiao, L.; Zhang, G.; Niu, B.; Chen, Z. Virtual Data Space System for National High-performance Computing Environment. *Big Data Res.* **2021**, *7*, 101–122.
11. He, X.; Deng, S.; Luan, H.; Niu, B. Study of Technique Support on the Operation of Virtual Data Space in National High Performance Computing Environment. *Big Data Res.* **2021**, *7*, 158–171.
12. Yu, Z.; Xiaodong, L.; Xinjian, Z. Distributed Data Uniform Access Technology Based on Metadata. *Command. Inf. Syst. Technol.* **2019**, *10*, 33–37.
13. Oracle rev.8.7. 2023. Available online: https://www.oracle.com/cn/database/ (accessed on 27 March 2024).
14. PostgreSQL rev.9.1. 2023. Available online: https://www.postgresql.org/ (accessed on 27 March 2024).
15. Corbett, J.C.; Dean, J.; Epstein, M.; Fikes, A.; Frost, C.; Furman, J.J.; Ghemawat, S.; Gubarev, A.; Heiser, C.; Hochschild, P.; et al. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst. (TOCS)* **2013**, *31*, 1–22. [CrossRef]
16. Huang, D.; Liu, Q.; Cui, Q.; Fang, Z.; Ma, X.; Xu, F.; Shen, L.; Tang, L.; Zhou, Y.; Huang, M.; et al. TiDB: A Raft-based HTAP database. *Proc. VLDB Endow.* **2020**, *13*, 3072–3084. [CrossRef]
17. Taft, R.; Sharif, I.; Matei, A.; VanBenschoten, N.; Lewis, J.; Grieger, T.; Niemi, K.; Woods, A.; Birzin, A.; Poss, R.; et al. Cockroachdb: The resilient geo-distributed sql database. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 1493–1509.
18. Yang, Z.; Yang, C.; Han, F.; Zhuang, M.; Yang, B.; Yang, Z.; Cheng, X.; Zhao, Y.; Shi, W.; Xi, H.; et al. OceanBase: A 707 million tpmC distributed relational database system. *Proc. VLDB Endow.* **2022**, *15*, 3385–3397. [CrossRef]
19. Cao, W.; Li, F.; Huang, G.; Lou, J.; Zhao, J.; He, D.; Sun, M.; Zhang, Y.; Wang, S.; Wu, X.; et al. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In Proceedings of the 2022 IEEE 38th International Conference on Data Engineering (ICDE), Kuala Lumpur, Malaysia, 9–12 May 2022; pp. 2859–2872.
20. Koutroumanis, N.; Nikitopoulos, P.; Vlachou, A.; Doulkeridis, C. NoDA: Unified NoSQL data access operators for mobility data. In Proceedings of the 16th International Symposium on Spatial and Temporal Databases, Vienna, Austria, 19–21 August 2019; pp. 174–177.
21. Apache Drill rev.3.3.4. 2023. Available online: https://drill.apache.org/ (accessed on 27 March 2024).
22. Shvachko, K.; Kuang, H.; Radia, S.; Chansler, R. The Hadoop Distributed File System. In Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, USA, 3–7 May 2010; pp. 1–10. [CrossRef]
23. Apache HBase rev.3.3.4. 2023. Available online: https://hbase.apache.org/ (accessed on 27 March 2024).

24. Quicksql rev.0.7.0. 2023. Available online: https://github.com/qihoo360/Quicksql/ (accessed on 27 March 2024).
25. SuperSQL rev.0.1. 2023. Available online: https://www.tencentcloud.com/document/product/1155/54462 (accessed on 27 March 2024).
26. Huo, J.; Xiao, L.; Huo, Z.; Xu, Y. Research and Implementation of Edge Cache System in Global Virtual Data Space Across WAN. *Big Data Res.* **2021**, *7*, 57–81.
27. Tan, H.; Jiang, S.H.C.; Han, Z.; Liu, L.; Han, K.; Zhao, Q. Camul: Online Caching on Multiple Caches with Relaying and Bypassing. In Proceedings of the IEEE Conference on Computer Communications, Paris, France, 29 April–2 May 2019; pp. 244–252.
28. Binqiang, C.; Chenyang, Y.; Gang, W. High throughput opportunistic cooperative device-to-device communications with caching. *IEEE Trans. Veh. Technol.* **2017**, *66*, 7527–7539.
29. AWS Cloud Services-Professional Big Data and Cloud Computing Services and Cloud Solution Providers rev.2023. 2023. Available online: https://aws.amazon.com/ (accessed on 27 March 2024).
30. Cloud Computing Services | Microsoft Azure rev.2023. 2023. Available online: https://azure.microsoft.com/ (accessed on 27 March 2024).
31. Aliyun-For Incalculable Value rev.2023. 2023. Available online: https://www.aliyun.com/ (accessed on 27 March 2024).
32. DB-Engines Ranking. Available online: https://db-engines.com/en/ranking (accessed on 27 March 2024).
33. Apache JMeter-Apache JMeter™. Available online: https://jmeter.apache.org/ (accessed on 27 March 2024).